



Segurança de Dados em SQL: Usuários, Autorizações e Views

INTRODUÇÃO

Segurança de dados é um tópico importante nos dias de hoje. Administradores de bancos de dados devem garantir que os dados sejam acessados apenas por pessoas e programas aplicativos autorizados. Os conceitos e recursos de segurança de dados em SQL dizem respeito unicamente ao acesso aos dados armazenados. Não é função dos SGBDR proteger a infraestrutura tecnológica (rede, servidores etc.) contra ataques e invasões.

Em SQL, segurança de dados é baseada em 3 conceitos:

- **Usuários.** São aqueles que utilizam o banco de dados. Qualquer ação sobre os objetos de um banco de dados (ver a seguir) tem associado um usuário. O RDBMS permite ou não ações dependendo do usuário responsável;

- **Objetos do banco de dados.** São os itens sobre os quais regras de segurança são aplicadas. Normalmente, regras de segurança são aplicadas a tabelas e *views*, porém podem ser estendidas a outros objetos como programas armazenados ou até a todo o banco de dados;

- **Permissões (ou Privilégios ou Autorizações).** Conjunto de ações que um usuário tem permissão para executar sobre um conjunto de objetos do banco de dados. As permissões mais comuns são relacionadas aos comandos SELECT, INSERT, UPDATE e DELETE



aplicados a um conjunto de tabelas. Pode-se também estabelecer permissões para outros comandos, como CREATE, ALTER e DR...

Um usuário é identificado através de seu *user id* e uma senha, mas a forma como isto é implementado varia de acordo com o SGBDR utilizado. A maioria dos SGBDR comerciais utiliza o conceito de **sessão** para identificar o usuário. Uma sessão é iniciada quando o usuário se conecta ao SGBDR, seja através de uma rede ou localmente, caso ambos estejam utilizando mesmo servidor, e termina quando a conexão é encerrada. No MySQL, por exemplo, antes de acessar o SGDBR, o *front end* MySQL Workbench solicita ao usuário seu *user id* e senha. Após o usuário se identificar, o programa se conecta ao SGDBR com as permissões do usuário em questão.[1]

CRIAÇÃO DE USUÁRIOS

Os *scripts* SQL para criação das tabelas utilizadas nos exemplos encontram-se nos materiais de apoio.[2]

A forma de se criar usuários varia de acordo com o SGBDR. O comando CREATE USER do MySQL, com suas principais cláusulas, encontra-se abaixo:

```
CREATE USER [IF NOT EXISTS]
    usuário1 [IDENTIFIED BY 'senha1'] [, usuario2   [IDENTIFIED
BY 'senha2']] ...
    DEFAULT ROLE perfil1 [, perfil2] ...
    [REQUIRE {NONE | opção_segurança1 [[AND]
opção_segurança2] ...}]
    [WITH opção_recurso1 [opção_recurso2] ...]
```

A forma mais simples de se criar um novo usuário é mostrada abaixo. O usuário *contabilidade* com senha *Troqueasenha123* é criado,

porém sem qualquer autorização. O usuário não consegue criar esquemas ou tabelas.

```
CREATE USER contabilidade IDENTIFIED BY 'Troqueasenha123';
```

Para atribuir autorizações ao usuário, deve-se utilizar o comando GRANT, mostrado mais adiante. Um cuidado: os principais SGDBR comerciais são sensíveis a letras maiúsculas e minúsculas (*case sensitive*) para nome de usuário e senha, mas não para nomes de objetos.

As demais cláusulas têm suas funções explicadas a seguir:

- DEFAULT ROLE – Atribui ao usuários os perfis relacionados (perfis serão apresentados mais adiante);
- REQUIRE – Determina que formas de autenticação serão utilizadas para o usuário. O padrão é NONE;
- WITH – Define as quotas de recursos atribuídas ao usuário. As quotas que podem ser definidas são:
  - MAX_QUERIES_PER_HOUR – Número máximo de consultas no período de uma hora;
  - MAX_UPDATES_PER_HOUR – Número máximo de atualizações no período de uma hora;
  - MAX_CONNECTIONS_PER_HOUR – Número máximo de conexões estabelecidas no período de uma hora;
  - MAX_USER_CONNECTIONS – Número máximo de conexões simultâneas.

A quota de cada recurso é um número inteiro e positivo. Os exemplos a seguir ilustram a utilização do comando CREATE USER:

```
CREATE USER joao@abc.com.br IDENTIFIED BY '2020@';
CREATE USER jose@localhost WITH MAX_USER_CONNECTIONS 1 MAX_QUERIES_PER_HOUR 20;
```

O primeiro comando cria o usuário *joao@abc.com.br* com senha '2020@'. O segundo, cria o usuário *jose*, com permissão de acesso local, sem uso de senha e com limitações no número de conexões simultâneas (1) e número máximo de consultas no período de uma hora (20).

ATRIBUIÇÃO DE AUTORIZAÇÕES

Autorizações permitem que usuários manipulem objetos e executem comandos SQL. O administrador do SGDBR possui o nível máximo de autorização, podendo atribuir ou alterar os privilégios de qualquer usuário. Na sua forma mais geral, o comando GRANT[3] atribui a usuários permissão para executar um conjunto de comandos sobre um conjunto de objetos:

```
GRANT {comando | ALL} [(colunas)] [, {comando | ALL} [(colunas)]] ...
ON [tipo_objeto] autorizações TO usuário_ou_perfil [, usuário_ou_perfil] ...
[WITH GRANT OPTION]
```

O modificador ALL atribui ao usuário todas as autorizações disponíveis, podendo até atribuir GRANT ALL a outro usuário caso a cláusula WITH GRANT OPTION esteja presente. Para autorizar um conjunto de comandos no mesmo comando GRANT, devem-se separá-los por vírgulas. O que pode ser autorizado também depende do SGBDR utilizado. No MySQL existem algumas dezenas de opções. As mais comuns, presentes em todos os principais SGBDR, são: CREATE, ALTER, DROP, SELECT, INSERT, UPDATE e DELETE.

Para comandos que manipulam colunas (SELECT, ALTER TABLE, INSERT e UPDATE) é possível fornecer a lista de colunas que podem ser manipuladas.[4] A cláusula ON especifica para que objetos estão sendo dadas as autorizações, podendo ser TABLE, FUNCTION ou PROCEDURE. TABLE deve ser utilizada se as autorizações são para uma tabela apenas. Os dois últimos tipos referem-se à programação SQL e não serão vistos aqui. Finalmente, a cláusula TO especifica o usuário que recebe as autorizações.

Exemplos:

```
GRANT SELECT (a1, a2) ON X.a TO usuario1;
GRANT SELECT, CREATE, UPDATE, DELETE, INSERT ON X.* TO usuario1;
GRANT ALL ON *.* TO usuario1;
GRANT SELECT ON TABLE x TO usuario1;
FLUSH PRIVILEGES;
```

O primeiro exemplo atribui ao *usuario1* privilégios de consulta (SELECT) das colunas a1 e *a2*, da tabela *a*, do esquema *X*. No segundo, é dada autorização para execução dos 5 comandos listados em qualquer tabela do esquema *X*. No terceiro, atribuem-se privilégios de administrados ao usuário1, exceto para o comando GRANT ALL. Finalmente, no quarto exemplo, usuário1 pode apenas consultar a tabela *a* no esquema selecionado.[5]

O comando FLUSH PRIVILEGES faz com que o SGDBR leia as tabelas de sistema e recarregue todas as informações de autorizações, atribuindo-as aos respectivos usuários ativos. Recomenda-se o seu uso após os comandos GRANT e REVOKE.

CRIAÇÃO E ATRIBUIÇÃO DE PERFIS

Atribuir autorizações a cada novo usuário criado pode ser um trabalho repetitivo e sujeito a erros. Na maioria dos casos, os usuários cadastrados recebem conjuntos de autorizações que tendem a se repetir. A linguagem SQL permite que sejam criados e nomeados conjuntos de autorizações padrão, chamados de perfis (*roles*). O comando CREATE ROLE cria perfis:

```
CREATE ROLE [IF NOT EXISTS] nome_perfil1 [, nome_perfil2 ] ...
```

Em um mesmo comando, podem ser criados vários perfis, porém todos "nascem" vazios. O comando GRANT é utilizado para se atribuir autorizações a perfis existentes, bastando substituir o nome do usuário pelo nome do perfil. No exemplo abaixo, dois perfis são criados e suas respectivas autorizações são atribuídas a eles:

```
CREATE ROLE UsuarioExterno, UsuarioInterno;
    GRANT SELECT ON AcessoExterno.* TO UsuarioExterno;
    GRANT SELECT, CREATE, INSERT, ALTER, DELETE ON AcessoInterno.* TO UsuarioInterno;
    FLUSH PRIVILEGES;
```

Os perfis criados podem ser atribuídos aos novos usuários:

```
CREATE USER joao@abc.com.br IDENTIFIED BY '2020@'
DEFAULT ROLE UsuarioExterno;
FLUSH PRIVILEGES;
```



A partir do momento em que *joao@abc.com.br* estabelecer uma conexão com o SGBDR, as autorizações definidas em *UsuarioExterno* passam a valer.

Perfis também podem ser atribuídos a usuários existentes. Assumindo que o usuário *luiz@localhost* já tenha sido criado:

```
SET DEFAULT ROLE UsuarioInterno TO luiz@localhost;
```

O comando REVOKE revoga autorizações atribuídas ao usuário. Há duas formas para o comando: revogação de comandos e revogação de perfis. Os exemplos abaixo ilustram as duas formas:

```
REVOKE CREATE, UPDATE, DELETE, INSERT ON X.* TO
usuario1;
REVOKE UsuarioInterno FROM luiz@localhost;
FLUSH PRIVILEGES;
```

VISÕES (ou *VIEWS*)

Views são tabelas virtuais, criadas a partir de consultas (consultas de criação ou de origem) que ficam armazenadas em um esquema. Mesmo sendo virtuais, elas se mostram aos usuários como uma tabela qualquer. Uma *view* possui nome, colunas e linhas, porém não fica armazenada de forma permanente no SGBDR. Quando uma consulta faz referência a uma *view*, o SGBDR executa a consulta de criação, armazena o resultado na memória do servidor e utiliza este resultado como se fosse uma tabela normal.[1] Praticamente tudo o

que pode ser feito com uma tabela normal pode também ser feito com uma *view*.

Vantagens no uso de *views*

Em aplicações comerciais, o uso de *views* apresenta inúmeras vantagens, principalmente na customização do banco de dados para diferentes públicos e no controle de acesso a dados. Suas principais vantagens são:

- **Segurança.** A atribuição de perfis a usuários restringe o acesso deles a objetos ou comandos apenas. *Views* podem segmentar os dados tanto verticalmente (colunas) como horizontalmente (linhas) fazendo com que cada usuário tenha acesso a um conjunto de *views* com aquilo que lhe é permitido ver;

- **Simplificação de consultas.** Consultas envolvendo muitas tabelas podem ser transformadas em *views*, expondo ao usuário apenas o resultado final;

- **Customização.** Diretores Executivos precisam ter uma visão geral e consolidada do negócio enquanto Gerentes de Divisões precisam de informações mais detalhadas. Muitas vezes, a informação é a mesma, mas com níveis de agregação distintos. *Views* podem (e devem) ser utilizadas para customizar a visão dos dados para diferentes públicos;

- **Interface padrão.** Imagine uma empresa que atue em um mercado muito dinâmico, em que as informações utilizadas para tomada de decisão mudem constantemente. Ou então um mercado muito regulado, em que as exigências legais também mudam com frequência. A manutenção de bancos de dados em ambientes como estes não é uma tarefa fácil. Com o uso de *views*, a camada

de exibição de dados não precisa ser alterada a cada alteração na estrutura das tabelas do banco de dados, tornando o trabalho de manutenção menos custoso e menos propenso a erros;

- **Integridade de dados.** Uitlizando *views* como interface para consulta e entrada de dados permite que o SGBDR verifique cada alteração feita e garanta a integridade dos dados.

Embora as vantagens sejam enormes, o uso de *views* apresenta algumas desvantagens:

- **Desempenho.** *Views* cujas consultas de criação são simples, com um número relativamente reduzido de tabelas de origem, são facilmente traduzidas e executadas sem a necessidade de criação de tabelas temporárias. Nos casos em que isto não ocorre, a consulta de criação deve ser executada e armazenada em uma tabela temporária, utilizando recursos do servidor;

- **Gerenciamento.** O uso indiscriminado de *views* pode causar mais problemas do que resolver. *Views* desenvolvidas sem padrões podem transformar a vida do administrados de banco de dados num caos. Como uma *view* é vista como uma tabela normal, *views* podem fazer referência a outras *views*, que por sua vez fazem referência a outras *views* e assim por diante. Quanto mais camadas entre uma *view* e as tabelas de origem, mais difícil fica sua manutenção e solução de problemas;

- **Atualizações.** Quando as linhas de uma *view* são alteradas, o SGBDR tenta mapear estas alterações diretamente em alterações nas tabelas de origem. Dependendo da complexidade da consulta de origem da *view*, o SGBDR não consegue resolver o mapeamento. Nestes casos, o SGBDR impede atualizações (*read only view*).



CRIAÇÃO E EXCLUSÃO DE *VIEWS*

Views são criadas a partir de consultas às tabelas de origem. Como exemplo, deseja-se criar duas *views*, separando os representantes das regiões leste e oeste:

```
CREATE VIEW REP_LESTE AS
SELECT * FROM REPRESENTANTES
WHERE filial_id IN (11, 12, 13);

CREATE VIEW REP_OESTE AS
SELECT * FROM REPRESENTANTES
WHERE filial_id IN (21, 22);
```

As consultas às duas *views* são mostradas abaixo:

REP_LESTE

rep_id	nome	idade	filial_id	cargo	data_inicio	gerente	meta	vendas
101	Dan Roberts	45	12	Sales Rep	2004-10-20	104	300000.00	305673.00
103	Paul Cruz	29	12	Sales Rep	2005-03-01	104	275000.00	286775.00
104	Bob Smith	33	12	Sales Mgr	2005-05-19	106	200000.00	142594.00
105	Bill Adams	37	13	Sales Rep	2006-02-12	104	350000.00	367911.00
106	Sam Clark	52	11	VP Sales	2006-06-14		275000.00	299912.00
109	Mary Jones	31	11	Sales Rep	2007-10-12	106	300000.00	392725.00

REP_OESTE

rep_id	nome	idade	filial_id	cargo	data_inicio	gerente	meta	ven
102	Sue Smith	48	21	Sales Rep	2004-12-10	108	350000.00	474050.00
107	Nancy Angelli	49	22	Sales Rep	2006-11-14	108	300000.00	186042.00
108	Larry Fitch	62	21	Sales Mgr	2007-10-12	106	350000.00	361865.00

A *view* implementada acima é chamada de *view* horizontal, pois segmenta as tabelas de origem por linhas. Outra forma de segmentação é a vertical, onde apenas algumas colunas das tabelas de origem são reproduzidas na *view*. Por exemplo, deseja-se colocar à disposição dos representantes uma lista com nome, escritório e região de todos:

```
CREATE VIEW REP_INFO AS
        SELECT CONCAT(R.rep_id, ' - ', R.nome) AS NOME , F.cidade
        AS CIDADE,
        F.regiao AS REGIÃO
        FROM REPRESENTANTES AS R, FILIAIS AS F
        WHERE F.filial_id = R.filial_id;
```

REP_INFO

NOME	CIDADE	REGIÃO
101 - Dan Roberts	Chicago	Leste
102 - Sue Smith	Los Angeles	Oeste
103 - Paul Cruz	Chicago	Leste
104 - Bob Smith	Chicago	Leste
105 - Bill Adams	Atlanta	Leste
106 - Sam Clark	New York	Leste
107 - Nancy Angelli	Denver	Oeste
108 - Larry Fitch	Los Angeles	Oeste
109 - Mary Jones	New York	Leste

Views podem ter agrupamentos em suas consultas de origem. O exemplo abaixo sumariza os pedidos para cada representante em uma *view* e, posteriormente, uma consulta utiliza a *view* para exibir o resultado:

```
CREATE VIEW PEDIDOS_POR_REP (rep_id, qtd, total, min, med.
max) AS
  SELECT rep_id, COUNT(*), SUM(valor), MIN(valor), AVG(valor),
  MAX(valor),
  FROM PEDIDOS
  GROUP BY rep_id;

SELECT CONCAT(R.rep_id, ' - ', R.nome) AS NOME, P.qtd AS
  QTD,
  FORMAT(P.total, 2, 'de_DE') AS TOTAL, FORMAT(P.min, 2,
  'de_DE') AS 'MÍNIMO', FORMAT(P.med, 2, 'de_DE') AS 'MÉDIA',
  FORMAT(P.max, 2, 'de_DE') AS 'MÁXIMO'
FROM REPRESENTANTES AS R, PEDIDOS_POR_REP AS P
WHERE P.rep_id = R.rep_id
ORDER BY TOTAL DESC;
```

NOME	QTD	TOTAL	MÍNIMO	MÉDIA	MÁXIMO
108 - Larry Fitch	7	58.633,00	652,00	8.376,14	45.000,00
105 - Bill Adams	5	39.327,00	702,00	7.865,40	27.500,00
107 - Nancy Angelli	3	34.432,00	652,00	11.477,33	31.350,00
106 - Sam Clark	2	32.958,00	1.458,00	16.479,00	31.500,00
101 - Dan Roberts	3	26.628,00	150,00	8.876,00	22.500,00
110 - Tom Snyder	2	23.132,00	632,00	11.566,00	22.500,00
102 - Sue Smith	4	22.776,00	1.896,00	5.694,00	15.000,00
109 - Mary Jones	2	7.105,00	1.480,00	3.552,50	5.625,00
103 - Paul Cruz	2	2.700,00	600,00	1.350,00	2.100,00

Observe que foram atribuídos nomes às colunas da *view*. Isto sempre deve ser feito em *views* com agregação (GROUP BY).

A exclusão de *views* é feita através do comando DROP VIEW, como no exemplo abaixo:

```
DROP VIEW PEDIDOS_POR_REP;
```

ATUALIZAÇÃO DE *VIEWS*

Como em qualquer outra tabela, é possível incluir, excluir e alterar linhas de uma *view*. Tomando como exemplo a *view* REPR_LESTE criada anteriormente, pode-se fazer:

```
INSERT INTO REPR_LESTE (rep_id, nome, filial_id, idade, data_inicio, vendas)
VALUES (113, 'Jake Kimball', 11, 43, '2009-01-01', 0.00);

UPDATE REPR_LESTE SET idade = 34 WHERE rep_id = 113;

DELETE FROM REPR_LESTE WHERE rep_id = 113;
```

Como as linhas de REP_LESTE e REPRESENTANTES têm uma correspondência de 1 para 1, ambas são atualizadas pelos comandos acima. Já com a *view* PEDIDOS_POR_REP isto não ocorre, porque ela possui um agrupamento. A *view* REP_INFO pode ter as colunas CIDADE e REGIAO atualizadas, mas NOME não (NOME é uma coluna composta por duas colunas da tabela de origem). Para saber se uma *view* é modificável, as seguintes regras se aplicam:

- Não pode possuir agrupamento (funções de agrupamento e cláusula GROUP BY);
- Não pode possuir a cláusula DISTINCT;

- Não pode ser criada a partir de UNION;



- Não podem ter OUTER JOIN;
- Não podem ter na cláusula FROM *views* não modificáveis.

As regras acima são gerais, porém cada SGDBR pode restringir mais ou relaxar algumas das regras acima. Por exemplo, o MySQL permite a alteração de *views* com colunas calculadas, desde que estas colunas não sejam alteradas.

Imagine agora que um novo escritório será inaugurado na cidade de San Francisco, na região oeste.:

```
INSERT INTO FILIAIS (filial_id, cidade, regiao, meta, vendas)
VALUES (23, 'San Francisco', 'Oeste', NULL, 0, 0);
```

Um novo representante acabou de ser contratado para o novo escritório:

```
INSERT INTO REP_OESTE (rep_id, nome, filial_id, idade,
data_inicio, vendas)
VALUES (114, 'Fred Roberts', 23, 47, '2009-01-01', 0.00);
```

A inserção poderia ter sido feita tanto em REP_OESTE quanto em REPRESENTANTES. Agora, o Vice-Presidente de Vendas lista todos os representantes da região oeste ...

```
SELECT * FROM REP_OESTE;
```

... recebendo o resultado abaixo:

REP_OESTE

rep_id	nome	idade	filial_id	cargo	data_inicio	gerente	meta	vendas
102	Sue Smith	48	21	Sales Rep	2004-12-10	108	350000.00	474050.00
107	Nancy Angelli	49	22	Sales Rep	2006-11-14	108	300000.00	186042.00
108	Larry Fitch	62	21	Sales Mgr	2007-10-12	106	350000.00	361865.00

O que aconteceu com a linha recém-inserida em REP_OESTE? Na realidade, não aconteceu nada de mais. Ela foi inserida corretamente em REPRESENTANTE. Porém, quando o comando SELECT foi executado, o SGBDR verificou a consulta de geração da *view*, traduziu as referências e executou a consulta sobre REPRESENTANTES. Ocorre que a consulta de origem tem, em sua cláusula WHERE, a condição *filial_id* IN (21, 22). Como o novo representante foi vinculado à filial 23, ele não faz parte de REP_OESTE.

A linguagem SQL fornece um mecanismo que previne a ocorrência de situações como esta. Na criação de uma *view*, deve-se incluir a cláusula abaixo, após a consulta de origem:

```
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

A função desta cláusula é manter a coerência entre as diversas *views* e entre estas e as tabelas de origem. Inclusões ou alterações de linhas que não atendam aos critérios de selação da *view* (cláusula WHERE do comando SELECT) são rejeitados, caso a cláusula CHECK OPTION esteja presente. A cláusula CASCADED propaga a verificação para outras *views* que estejam na cláusula FROM da consulta de criação, de forma similar à cláusula CASCADE presente

na criação de chaves estrangeiras. O modificador LOCAL restringe a verificação à *view* sujeita à inclusão ou alteração.

Para evitar a ocorrência do problema visto acima, a *view* REP_OESTE deve ser definida como:

```
CREATE VIEW REPR_OESTE AS
  SELECT R.* FROM REPRESENTANTES AS R, FILIAIS AS F
  WHERE F.filial_id = R.filial_id AND F.Regiao = 'Oeste'
WITH LOCAL CHECK OPTION;
```

Verificações de coerência também podem ser especificadas no comando DROP VIEW. Para ilustrar o seu funcionamento, considere a *view* que relaciona os representantes vinculados à filial de Chicago (*filial_id* = 12):

```
CREATE VIEW REP_CHICAGO AS
  SELECT * FROM REP_LESTE
  WHERE filial_id = 12;
```

As cláusulas CASCADE e RESTRICT podem ser usadas em conjunto com DROP VIEW. CASCADE propaga o DROP VIEW para todas as *views* que utilizam a *view* excluída e RESTRICT impede que uma *view* referenciada por outras *views* seja excluída. O primeiro comando abaixo é rejeitado enquanto o segundo exclui tanto REP_LESTE quanto REP_CHICAGO[7]:

```
DROP VIEW REP_LESTE RESTRICT;
    DROP VIEW REP_LESTE CASCADE;
```

[1] Todo SGBDR, ao ser instalado, cria um superusuário que possui todas as autorizações disponíveis. No MySQL este usuário é chamado root. Autorizações podem ser dadas a usuários, domínios (@xxx.com, por exemplo) ou usuários em domínios (fulano@xxx.com). No sistema operacional Windows, quando cliente (programa que acessa o SGBDR) e servidor estão na mesma máquina, o domínio é identificado como '@localhost'

[2] Os exemplos utilizados no texto foram adaptados de GROFF, J. R., WEINBERG, P. N. e OPPEL, A. J. SQL: The Complete Reference. 3a Ed., Nova York: McGraw-Hill, 2009. Os scripts de criação e população das tabelas pode ser baixado gratuitamente em www.mhprofessional.com/computingdownload.

[3] Antes de utilizar o comando GRANT, deve-se verificar qual é o esquema em uso através do comando SELECT DATABASE().

[4] Seleção de colunas distintas para os comandos de manipulação pode causar confusão. Recomenda-se cautela.

[5] Pode-se definir um esquema padrão para a sessão ativa utilizando-se o comando USE nome_esquema, porém não há qualquer comando que desfaça o USE. Para deixar a sessão sem esquema padrão, deve-se usar o seguinte artifício: CREATE SCHEMA nome_qualquer; USE nome_qualquer; DROP nome_qualquer; onde nome_qualquer deve ser diferente de todos os esquemas ativos. Não havendo qualquer esquema em uso, deve-se qualificar todas as referências a tabelas, como no comando GRANT ... ON TABLE.

[6] Na prática, não é exatamente assim que os SGBDR gerenciam views. Dependendo da consulta de criação, pode ser muito custoso ter que repeti-la a cada referência feita. Quando possível, o SGBDR traduz as referências à view em referências às tabelas que são utilizadas para cria-la (tabelas de origem), as consultando diretamente. Em situações em que a tradução não é possível, o SGBDR executa de fato a consulta de criação da view e armazena o resultado em tabelas temporárias.

[7] O MySQL aceita as cláusulas CASCADE e RESTRICT, porém elas não têm qualquer efeito sobre o comando DROP VIEW.



Atividade extra

Nome da atividade: Ler alguns artigos sobre Stored Procedures

Leia alguns artigos dos links disponíveis abaixo (utilize o tradutor automático do browser se necessário) para entender melhor o conceito de Stored Procedure e tente implementar os exemplos disponíveis.

Links:

https://docs.oracle.com/cd/B28359_01/appdev.111/b28843/tdddg_procedures.h

https://www.devmedia.com.br/como-trabalhar-com-stored-procedures-e-cursores-no-oracle-sql-server-e-firebird-e-postgresql/33023

Referência Bibliográfica

ELMASRI, R. e NAVATHE, S. B. **Sistemas de Banco de Dados**. 7ª Ed., São Paulo: Pearson, 2011.

RAMAKRISHNAN, R., GEHRKE, J. **Sistemas de Gerenciamento de Banco de Dados.** 3ª Ed., São Paulo: McGraw-Hill, 2008.

CORONEL, C. e ROB, P. **Sistemas de Banco de Dados - Projeto, Implementação e Gerenciamento.** 1ª Ed., São Paulo: Cengage, 2010.

GROFF, J. R., WEINBERG, P. N. e OPPEL, A. J. **SQL: The Complete Reference.** 3ª Ed., Nova York: McGraw-Hill, 2009.



Ir para exercício